

Agent 智能体算法研究与模拟

算法本身的缺陷如:"过高估计"

本人在利用 Q-Learning 算法对此算法在路径自主学习过程进行模拟,利用 python 编写迷宫环境,用于智能体寻路过程的自学习,发现此算法在用于路径自学习过程中暴露出算法本身的缺陷.

利用 tensorflow numpy 等用于数学计算的 python 包支持算法的计算,与每一论次的迭代学习.

模拟构建三维场景,再现模拟算法在路径规划上强化学习,并根据此环境下算法所暴露出来的问题进行参数调整,实现

Q-Learning 算法问题: 由于它使用最优价值的行动替代了交互时使用的行动,所以在估计时有可能造成对干扰行动的"过高估计". 由于 Q-Learning 算法本质上采用了采样的方法,所以对于被采样的状态行动,Q-Learning 会对其"过高估计"而对于没有采样到的状态行动,

则无法享受"过高估计"的待遇,那么两者之间就可以产生比较大的差距,这些差距会造成算法的波动,在真实的应用场景中更是如此.

#-greedy 策略. 一开始策略以 100%的概率随机产生行动,随着训练的不断进行,这个概率将不断衰减,最终衰减至 10%,也就是说有 90%的概率执行当前最优策略,以探索为主的策略逐渐转变为以利用为主的策略,两者得到了很好的结合.

最优价值思想算法,将重点放在值函数上,通过交互序列的信息学

习价值模型,并通过价值模型更新策略,其中的思想和价值迭代法十分相似.

随着强化学习和深度学习的共同发展,基于 Q-Learning 的算法获得了很大的突破,甚至达到了专家水平.

当智能体感知环境运转的细节时,如何从环境的变化信息作出相应的动作,

当智能体不能感知环境状态时.

强化学习自身的特点:不断试错,也就是通过尝试与环境的交互,来解决策略评估的问题:

1. 如何得到环境序列
2. 如何使用序列进行评估

本课题旨在通过研究强化学习的算法, 研究智能体从环境到动作的实现过程,并将其算法在模拟过程中出现的问题进行调整

对于在特定场景中的模拟过程中,算法所表现的不足,通过调整参数,实现最佳的环境--动作迁移过程.

本课题将在智能体路径自主学习规划问题上利用 Q-Learning 算法,并在模拟实验过程中对此算法所表现的不足,通过调整参数,实现最佳的环境--动作迁移过程.

SARSA 算法和 Q-Learning 算法在公式上的不同,实际上这两种算法代表了两种策略评估的方式.分别是 On-Policy 和 Off-Policy.

On-Policy 对值函数的更新是完全依据交互序列进行的,我们在计算时认为,价值可以直接使用采样的序列估计得到;

Off-Policy 在更新值函数时并不完全遵循交互序列,而是选择来自其他策略的交互序列的子部分替换了原本的交互序列,

从算法的思想上来说,Q-Learning 的思想更复杂,它结合了子部分的最优价值,更像是结合了价值迭代的更新算法,

希望每一次都使用前面迭代累计的最优结果进行更新.

本课题利用强化学习的算法对 Agent 智能体在路径选择上,加以优化利用,实现智能体的路径自学习。

强化学习中最优价值算法中最经典的为 q-learning 算法, q-learning 是强化学习算法中值迭代的算法,

Q 即为 $Q(s,a)$ 就是在某一时刻的 s 状态下($s \in S$), 采取 a ($a \in A$)动作能够获得收益的期望,

环境会根据 agent 的动作反馈相应的回报 reward r , 所以算法的主要思想就是将 State 与 Action 构建成一张 Q-table 来存储 Q 值,

然后根据 Q 值来选取动作获得较大的收益。为了模拟智能体在路径规划上寻找最优路径,

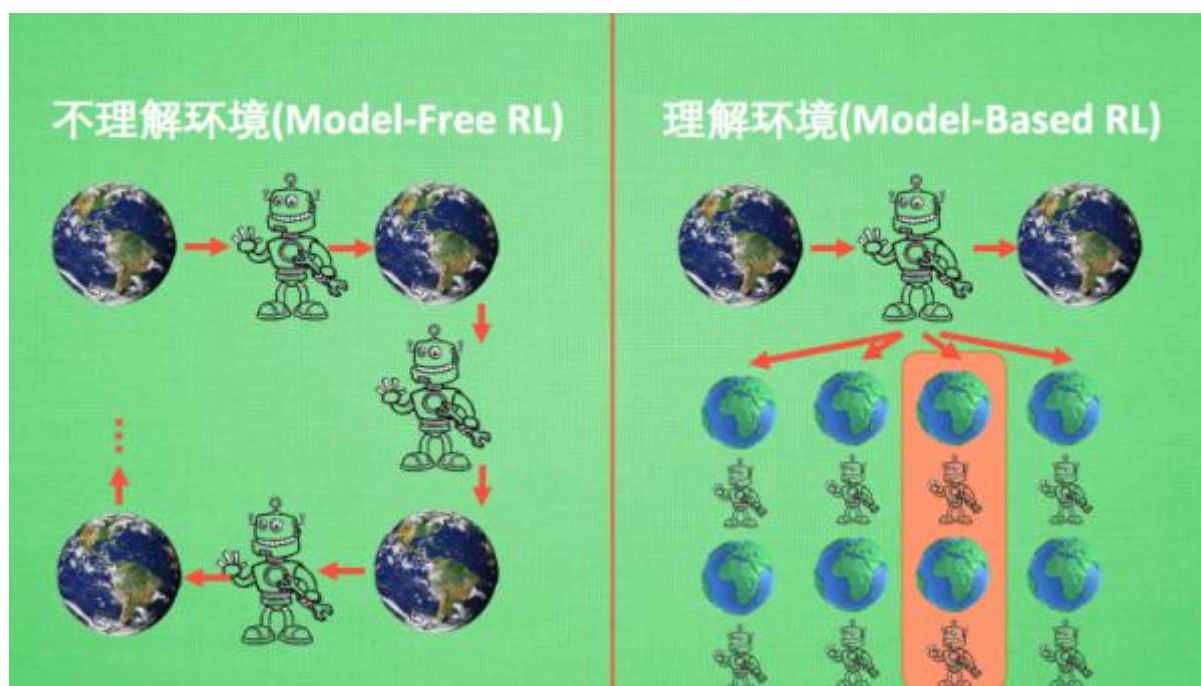
本课题通过选择设计应用场景,实现智能体在路径规划上的自学习功能.

没有任何 supervise(老师)的情况下学习如何做动作获得很高的分数,通过分数的导向性使 Agent 学习如何再去获得高分的动作.

强化学习方法汇总

了解强化学习中常用到的几种方法,以及他们的区别,对我们根据特定问题选择方法时很有帮助. 强化学习是一个大家族,发展历史也不短,具有很多种不同方法. 比如说比较知名的控制方法 **Q learning**, **Policy Gradients**, 还有基于对环境的理解的 model-based RL 等等. 接下来我们通过分类的方式来了解他们的区别.

Model-free 和 Model-based

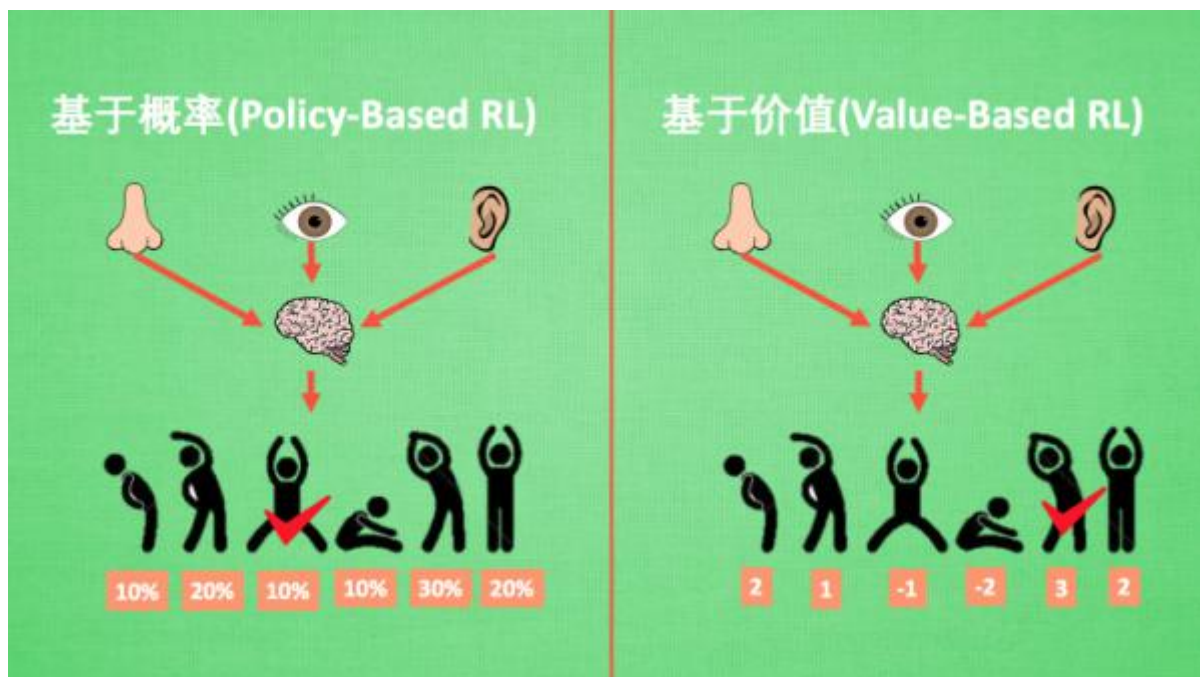


我们可以将所有强化学习的方法分为理不理解所处环境,如果我们不尝试去理解环境,环境给了我们什么就是什么. 我们就把这种方法叫做 model-free, 这里的 model 就是用模型来表示环境, 那理解了环境也就是学会了用一个模型来代表环境, 所以这种就是 model-based 方法. 我们想象. 现在环境就是我们的世界, 我们的机器人正在这个世界里玩耍, 他不理解这个世界是怎样构成的, 也不理解世界对于他的行为会怎么样反馈. 举个例子, 他决定丢颗原子弹去真实的世界, 结果把自己给炸死了, 所有结果都是那么现实. 不过如果采取的是 model-based RL, 机器人会通过过往的经验, 先理解真实世界是怎样的, 并建立一个模型来模拟现实世界的反馈, 最后他不仅可以在现实世界中玩耍,

也能在模拟的世界中玩耍，这样就没必要去炸真实世界，连自己也炸死了，他可以像玩游戏一样炸炸游戏里的世界，也保住了自己的小命。那我们就来说说这两种方式的强化学习各用那些方法吧。

Model-free 的方法有很多，像 **Q learning**, **Sarsa**, **Policy Gradients** 都是从环境中得到反馈然后从中学习。而 model-based RL 只是多了一道程序，为真实世界建模，也可以说他们都是 model-free 的强化学习，只是 model-based 多出了一个虚拟环境，我们不仅可以像 model-free 那样在现实中玩耍，还能在游戏中玩耍，而玩耍的方式也都是 model-free 中那些玩耍方式，最终 model-based 还有一个杀手锏是 model-free 超级羡慕的。那就是想象力。

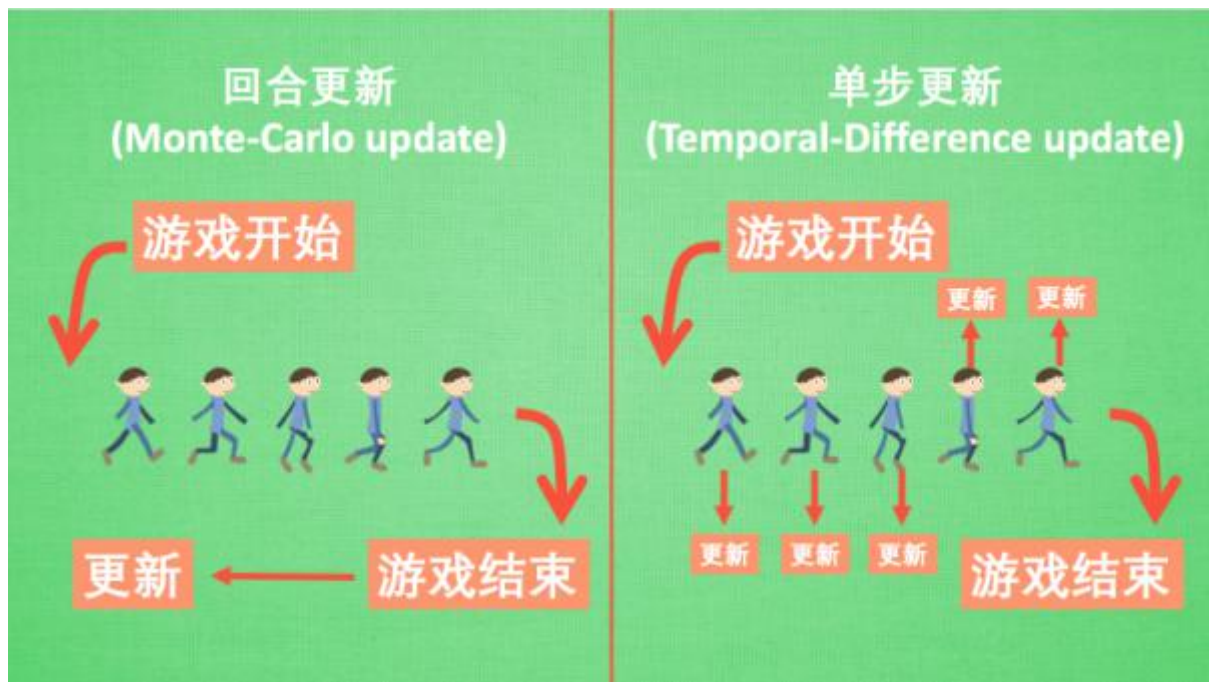
Model-free 中，机器人只能按部就班，一步一步等待真实世界的反馈，再根据反馈采取下一步行动。而 model-based，他能通过想象来预判断接下来将要发生的所有情况。然后选择这些想象情况中最好的那种。并依据这种情况来采取下一步的策略，这也就是围棋场上 AlphaGo 能够超越人类的原因。接下来，我们再来用另外一种分类方法将强化学习分为基于概率和基于价值。



基于概率是强化学习中最直接的一种，他能通过感官分析所处的环境，直接输出下一步要采取的各种动作的概率，然后根据概率采取行动，所以每种动作都有可能被选中，只是可能性不同。而基于价值的方法输出则是所有动作的价值，我们会根据最高价值来选着动作，相比基于概率的方法，基于价值的决策部分更为铁定，毫不留情，就选价值最高的，而基于概率的，即使某个动作的概率最高，但是还是不一定会选到他。

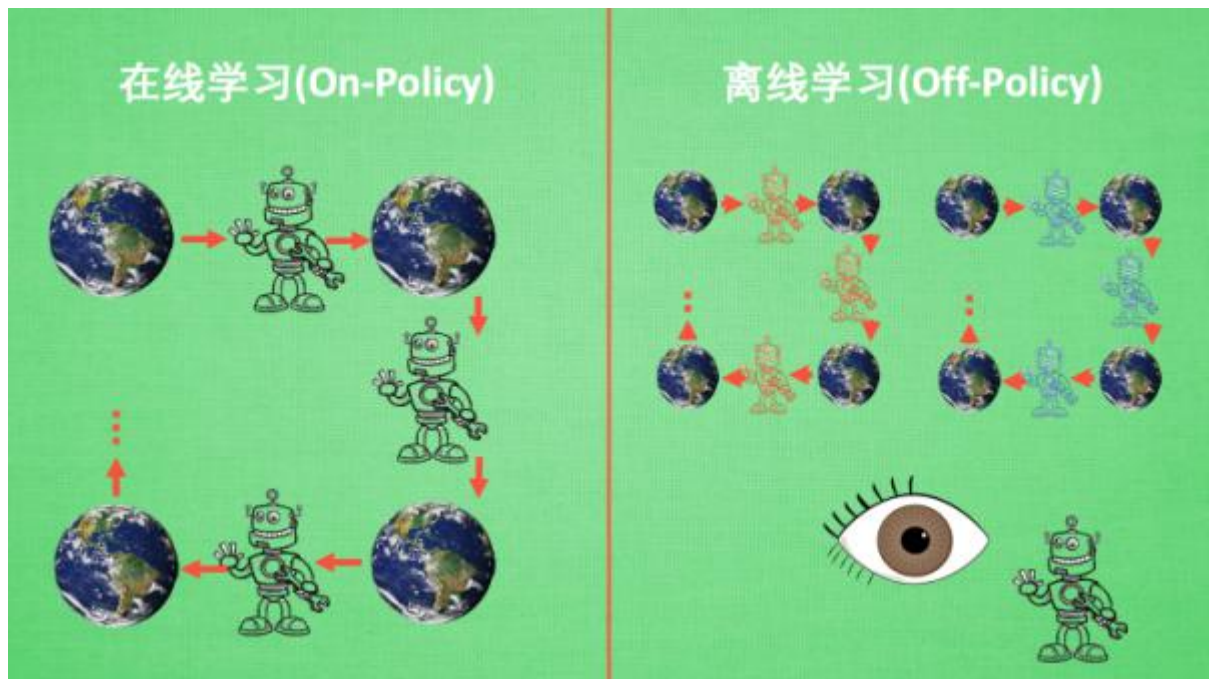
我们现在说的动作都是一个一个不连续的动作，而对于选取连续的动作，基于价值的方法是无能为力的。我们却能用一个概率分布在连续动作中选取特定动作，这也是基于概率的方法的优点之一。那么这两类使用的方法又有哪些呢？

比如在基于概率这边，有 **Policy Gradients**，在基于价值这边有 **Q learning**, **Sarsa** 等。而且我们还能结合这两类方法的优势之处，创造更牛逼的一种方法，叫做 **Actor-Critic**，actor 会基于概率做出动作，而 critic 会对做出的动作给出动作的价值，这样就在原有的 policy gradients 上加速了学习过程。



强化学习还能用另外一种方式分类，回合更新和单步更新，想象强化学习就是在玩游戏，游戏回合有开始和结束。回合更新指的是游戏开始后，我们要等待游戏结束，然后再总结这一回合中的所有转折点，再更新我们的行为准则。而单步更新则是在游戏进行中每一步都在更新，不用等待游戏的结束，这样我们就能边玩边学习了。

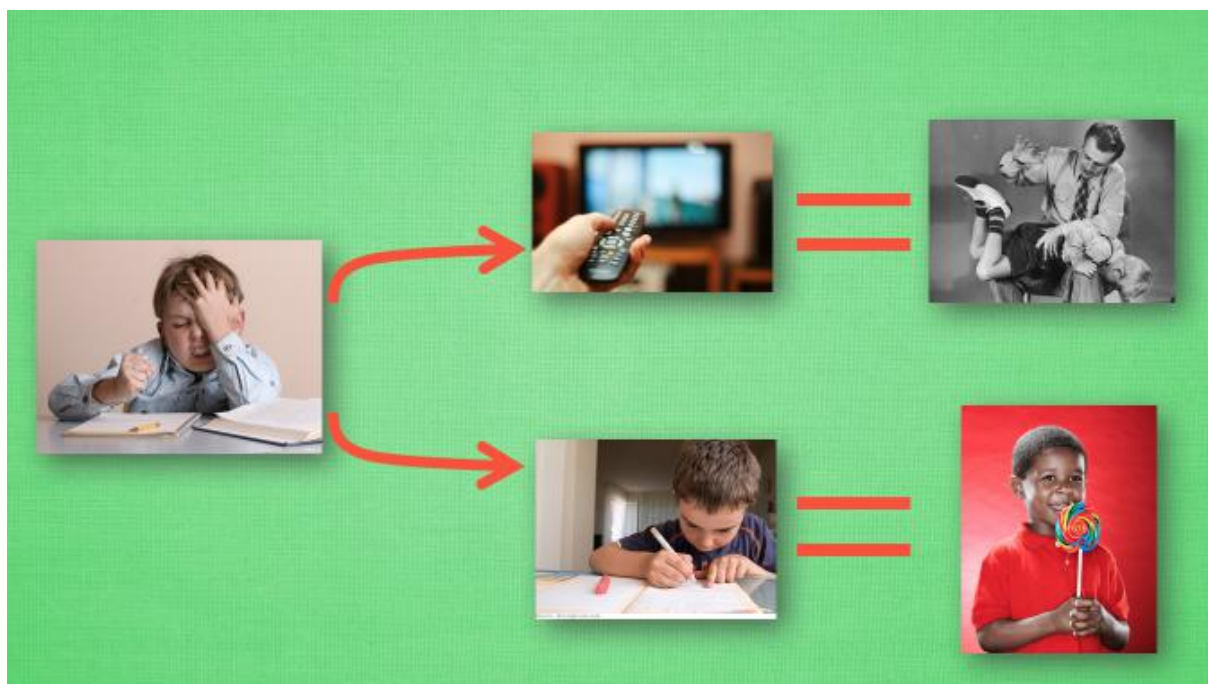
再来说说方法，Monte-carlo learning 和基础版的 policy gradients 等都是回合更新制，Qlearning, Sarsa, 升级版的 policy gradients 等都是单步更新制。因为单步更新更有效率，所以现在大多方法都是基于单步更新。比如有的强化学习问题并不属于回合问题。



所谓在线学习,就是指我必须本人在场,并且一定是本人边玩边学习,而离线学习是你可以选择自己玩,也可以选择看着别人玩,通过看别人玩来学习别人的行为准则,离线学习 同样是从过往的经验中学习,但是这些过往的经历没必要是自己的经历,任何人的经历都能被学习.或者我也不必要边玩边学习,我可以白天先存储下来玩耍时的记忆,然后晚上通过离线学习来学习白天的记忆.那么每种学习的方法又有哪些呢?

最典型的在线学习就是 Sarsa 了,还有一种优化 Sarsa 的算法,叫做 Sarsa lambda, 最典型的离线学习就是 Q learning, 后来人也根据离线学习的属性,开发了更强大的算法,比如让计算机学会玩电动的 Deep-Q-Network.

什么是 Q Learning



我们做事情都会有一个自己的行为准则，比如小时候爸妈常说“不写完作业就不准看电视”。所以我们在写作业的这种状态下，好的行为就是继续写作业，直到写完它，我们还可以得到奖励，不好的行为就是没写完就跑去看电视了，被爸妈发现，后果很严重。小时候这种事情做多了，也就变成我们不可磨灭的记忆。这和我们要提到的 Q learning 有什么关系呢？原来 Q learning 也是一个决策过程，和小时候的这种情况差不多。我们举例说明。

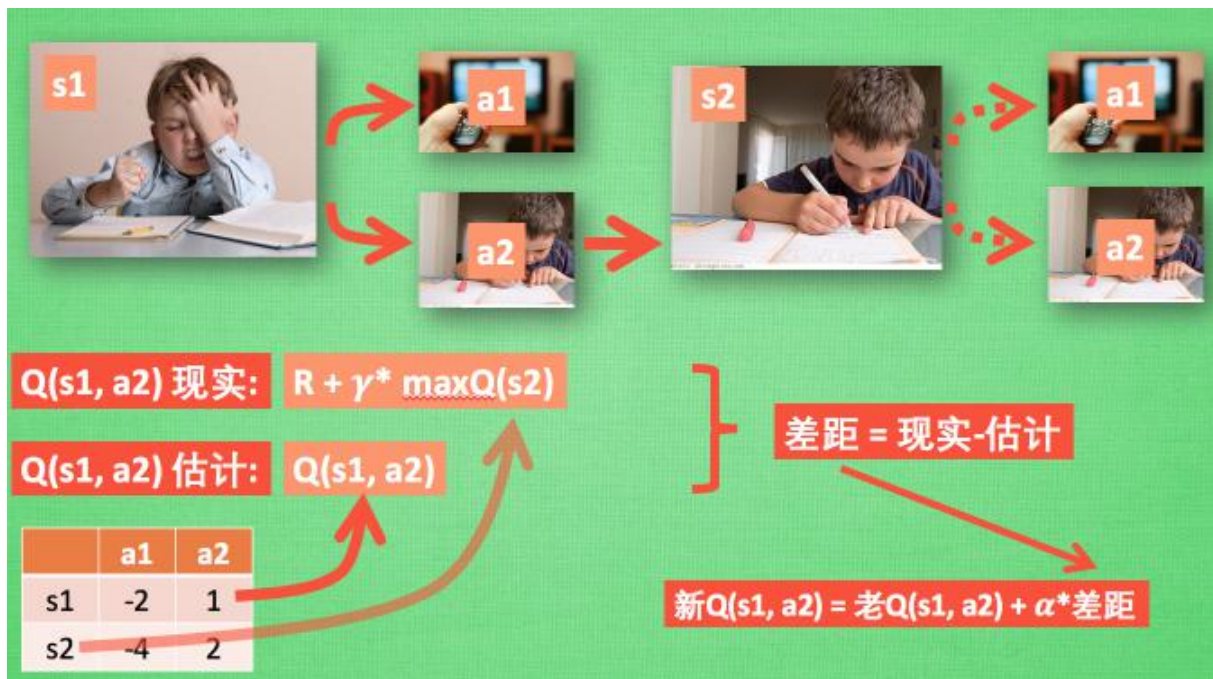
假设现在我们处于写作业的状态而且我们以前并没有尝试过写作业时看电视，所以现在我们有两种选择，1，继续写作业，2，跑去看电视。因为以前没有被罚过，所以我选看电视，然后现在的状态变成了看电视，我又选了继续看电视，接着我还是看电视，最后爸妈回家，发现我没写完作业就去看电视了，狠狠地惩罚了我一次，我也深刻地记下了这一次经历，并在我的脑海中将“没写完作业就看电视”这种行为更改为负面行为，我们在看看 Q learning 根据很多这样的经历是如何来决策的吧。

Q-Learning 决策 ¶



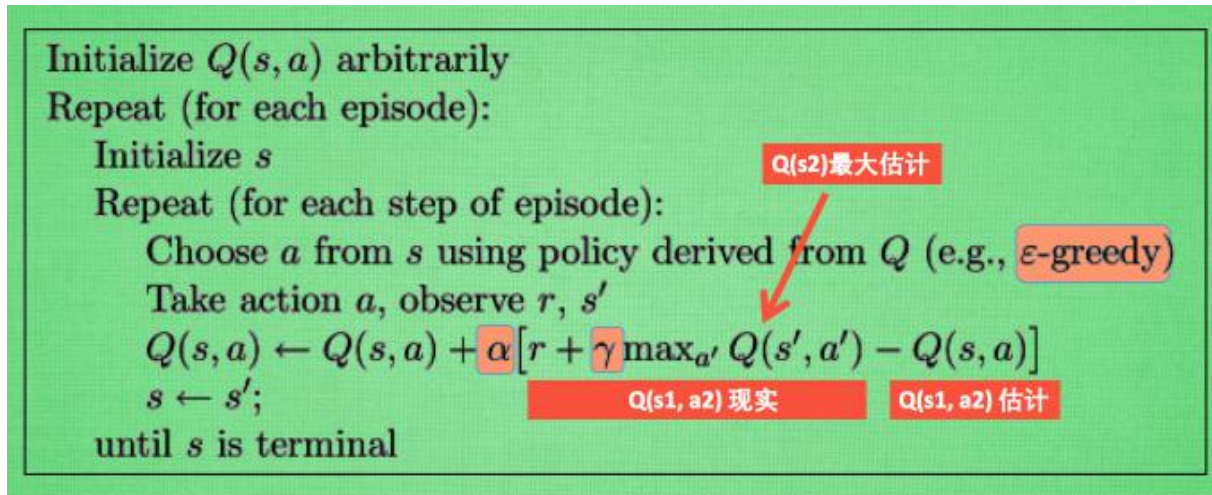
假设我们的行为准则已经学习好了,现在我们处于状态 s_1 ,我在写作业,我有两个行为 a_1, a_2 , 分别是看电视和写作业,根据我的经验,在这种 s_1 状态下, a_2 写作业带来的潜在奖励要比 a_1 看电视高,这里的潜在奖励我们可以用一个有关于 s 和 a 的 Q 表格代替,在我的记忆 Q 表格中, $Q(s_1, a_1) = -2$ 要小于 $Q(s_1, a_2) = 1$,所以我们判断要选择 a_2 作为下一个行为. 现在我们的状态更新成 s_2 ,我们还是有两个同样的选择,重复上面的过程,在行为准则 Q 表中寻找 $Q(s_2, a_1)$ $Q(s_2, a_2)$ 的值,并比较他们的大小,选取较大的一个. 接着根据 a_2 我们到达 s_3 并在此重复上面的决策过程. Q learning 的方法也就是这样决策的. 看完决策,我看来研究一下这张行为准则 Q 表是通过什么样的方式更改,提升的.

Q-Learning 更新



所以我们回到之前的流程, 根据 Q 表的估计, 因为在 s1 中, a2 的值比较大, 通过之前的决策方法, 我们在 s1 采取了 a2, 并到达 s2, 这时我们开始更新用于决策的 Q 表, 接着我们并没有在实际中采取任何行为, 而是再想象自己在 s2 上采取了每种行为, 分别看看两种行为哪一个的 Q 值大, 比如说 $Q(s2, a2)$ 的值比 $Q(s2, a1)$ 的大, 所以我们把大的 $Q(s2, a2)$ 乘上一个衰减值 gamma (比如是 0.9) 并加上到达 s2 时所获取的奖励 R (这里还没有获取到我们的棒棒糖, 所以奖励为 0), 因为会获取实实在在的奖励 R, 我们将这个作为我现实中 $Q(s1, a2)$ 的值, 但是我们之前是根据 Q 表估计 $Q(s1, a2)$ 的值. 所以有了现实和估计值, 我们就能更新 $Q(s1, a2)$, 根据 估计与现实的差距, 将这个差距乘以一个学习效率 alpha 累加上老的 $Q(s1, a2)$ 的值 变成新的值. 但时刻记住, 我们虽然用 $\max Q(s2)$ 估算了一下 s2 状态, 但还没有在 s2 做出任何的行为, s2 的行为决策要等到更新完了以后再重新另外做. 这就是 off-policy 的 Q learning 是如何决策和学习优化决策的过程.

Q-Learning 整体算法 ¶





这一张图概括了我们之前所有的内容. 这也是 Q learning 的算法, 每次更新我们都用到了 Q 现实和 Q 估计, 而且 Q learning 的迷人之处就是在 $Q(s1, a2)$ 现实中, 也包含了一个 $Q(s2)$ 的最大估计值, 将对下一步的衰减的最大估计和当前所得到的奖励当成这一步的现实, 很奇妙吧. 最后我们来说说这套算法中一些参数的意义. Epsilon greedy 是在决策上的一种策略, 比如 $\epsilon = 0.9$ 时, 就说明有 90% 的情况我会按照 Q 表的最优值选择行为, 10% 的时间使用随机选行为. α 是学习率, 来决定这次的误差有多少是要被学习的, α 是一个小于 1 的数. γ 是对未来 reward 的衰减值. 我们可以这样想象.


Q-Learning 中的 Gamma ¶

$$Q(s_1) = r_2 + \gamma Q(s_2) = r_2 + \gamma [r_3 + \gamma Q(s_3)] = r_2 + \gamma[r_3 + \gamma[r_4 + \gamma Q(s_4)]] = \dots$$

$$Q(s_1) = r_2 + \gamma r_3 + \gamma^2 r_4 + \gamma^3 r_5 + \gamma^4 r_6 + \dots$$

$\gamma = 1$

 $Q(s_1) = r_2 + 1 \cdot r_3 + 1 \cdot r_4 + 1 \cdot r_5 + 1 \cdot r_6 + \dots$

$\gamma = (0 \sim 1)$

 $Q(s_1) = r_2 + \gamma r_3 + \gamma^2 r_4 + \gamma^3 r_5 + \gamma^4 r_6 + \dots$

$\gamma = 0$

 $Q(s_1) = r_2$

我们重写一下 $Q(s_1)$ 的公式, 将 $Q(s_2)$ 拆开, 因为 $Q(s_2)$ 可以像 $Q(s_1)$ 一样, 是关于 $Q(s_3)$ 的, 所以可以写成这样, 然后以此类推, 不停地这样写下去, 最后就能写成这样, 可以看出 $Q(s_1)$ 是有关于之后所有的奖励, 但这些奖励正在衰减, 离 s_1 越远的状态衰减越严重. 不好理解? 行, 我们想象 Qlearning 的机器人天生近视眼, $\gamma = 1$ 时, 机器人有了一副合适的眼睛, 在 s_1 看到的 Q 是未来没有任何衰变的奖励, 也就是机器人能清清楚楚地看到之后所有步的全部价值, 但是当 $\gamma = 0$, 近视机器人没了眼镜, 只能摸到眼前的 reward, 同样也就只在乎最近的大奖励, 如果 γ 从 0 变到 1, 眼镜的度数由浅变深, 对远处的价值看得越清楚, 所以机器人渐渐变得有远见, 不仅仅只看眼前的利益, 也为自己的未来着想.

```

Initialize  $Q(s, a)$  arbitrarily
Repeat (for each episode):
    Initialize  $s$ 
    Repeat (for each step of episode):
        Choose  $a$  from  $s$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)
        Take action  $a$ , observe  $r, s'$ 
         $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$ 
         $s \leftarrow s'$ 
    until  $s$  is terminal

```

整个算法就是一直不断更新 Q table 里的值, 然后再根据新的值来判断要在某个 state 采取怎样的 action. Qlearning 是一个 off-policy 的算法, 因为里面的 **max** action 让 Q table 的更新可以不基于正在经历的经验(可以是现在学习着很久以前的经验, 甚至是学习他人的经验). 不过这一次的例子, 我们没有运用到 off-policy, 而是把 Qlearning 用在了 on-policy 上, 也就是现学现卖, 将现在经历的直接当场学习并运用. On-policy 和 off-policy 的差别我们会在之后的 **Deep Q network (off-policy)** 学习中见识到. 而之后的教程也会讲到一个 on-policy (Sarsa) 的形式, 我们之后再对比.

算法的代码形式 ¶

首先我们先 import 两个模块, **maze_env** 是我们的环境模块, 已经编写好了, 大家可以直接在[这里下载](#), **maze_env** 模块我们可以不深入研究, 如果你对编辑环境感兴趣, 可以去看看如何使用 python 自带的简单 GUI 模块 **tkinter** 来编写虚拟环境. 我也有[对应的教程](#). **maze_env** 就是用 **tkinter** 编写的. 而 **RL_brain** 这个模块是 RL 的大脑部分, 我们下节会讲.

```

from maze_env import Maze
from RL_brain import QLearningTable

```


下面的代码, 我们可以根据上面的图片中的算法对应起来, 这就是整个 Qlearning 最重要的迭代更新部分啦.

```
def update():
    # 学习 100 回合
    for episode in range(100):
        # 初始化 state 的观测值
        observation = env.reset()

        while True:
            # 更新可视化环境
            env.render()

            # RL 大脑根据 state 的观测值挑选 action
            action = RL.choose_action(str(observation))

            # 探索者在环境中实施这个 action, 并得到环境返回的下一个 state 观测值, reward
            # 和 done (是否是掉下地狱或者升上天堂)
            observation_, reward, done = env.step(action)

            # RL 从这个序列 (state, action, reward, state_) 中学习
            RL.learn(str(observation), action, reward, str(observation_))

            # 将下一个 state 的值传到下一次循环
            observation = observation_

            # 如果掉下地狱或者升上天堂, 这回合就结束了
            if done:
                break

        # 结束游戏并关闭窗口
        print('game over')
        env.destroy()

if __name__ == "__main__":
    # 定义环境 env 和 RL 方式
    env = Maze()
    RL = QLearningTable(actions=list(range(env.n_actions)))

    # 开始可视化环境 env
    env.after(100, update)
```

```
env.mainloop()
```

Q-learning 思维决策

代码主结构

与上回不一样的地方是, 我们将要以一个 class 形式定义 Q learning, 并把这种 tabular q learning 方法叫做 **QLearningTable**.

```
class QLearningTable:
    # 初始化
    def __init__(self, actions, learning_rate=0.01, reward_decay=0.9, e_greedy=0.9):

    # 选行为
    def choose_action(self, observation):

    # 学习更新参数
    def learn(self, s, a, r, s_):

    # 检测 state 是否存在
    def check_state_exist(self, state):
```

预设值

初始的参数意义不会在这里提及了, 请参考这个快速了解通道 [机器学习系列-Q learning](#)

```
import numpy as np
import pandas as pd

class QLearningTable:
    def __init__(self, actions, learning_rate=0.01, reward_decay=0.9, e_greedy=0.9):
        self.actions = actions # a list
        self.lr = learning_rate # 学习率
        self.gamma = reward_decay # 奖励衰减
        self.epsilon = e_greedy # 贪婪度
```

```
self.q_table = pd.DataFrame(columns=self.actions, dtype=np.float64) # 初始
q_table
```

决定行为 ¶

这里是定义如何根据所在的 state, 或者是在这个 state 上的 观测值 (observation) 来决策.

```
def choose_action(self, observation):
    self.check_state_exist(observation) # 检测本 state 是否存在(见后面标
    题内容)

    # 选择 action
    if np.random.uniform() < self.epsilon: # 选择 Q value 最高的 action
        state_action = self.q_table.loc[observation, :]

        # 同一个 state, 可能会有多个相同的 Q action value, 所以我们乱序一下
        action = np.random.choice(state_action[state_action ==
        np.max(state_action)].index)

    else: # 随机选择 action
        action = np.random.choice(self.actions)

    return action
```

学习 ¶

同上一个简单的 q learning 例子一样, 我们根据是否是 terminal state (回合终止符) 来判断应该如何更新 q_table. 更新的方式是不是很熟悉呢:

```
update = self.lr * (q_target - q_predict)
```

这可以理解成神经网络中的更新方式, 学习率 * (真实值 - 预测值). 将判断误差传递回去, 有着和神经网络更新的异曲同工之处.

```
def learn(self, s, a, r, s_):
```

```

self.check_state_exist(s_) # 检测 q_table 中是否存在 s_ (见后面标题
内容)
q_predict = self.q_table.loc[s, a]
if s_ != 'terminal':
    q_target = r + self.gamma * self.q_table.loc[s_, :].max() # 下个
state 不是 终止符
else:
    q_target = r # 下个 state 是终止符
self.q_table.loc[s, a] += self.lr * (q_target - q_predict) # 更新对应的
state-action 值

```

检测 state 是否存在 ¶

这个功能就是检测 `q_table` 中有没有当前 state 的步骤了, 如果还没有当前 state, 那我们就插入一组全 0 数据, 当做这个 state 的所有 action 初始 values.

```

def check_state_exist(self, state):
    if state not in self.q_table.index:
        # append new state to q table
        self.q_table = self.q_table.append(
            pd.Series(
                [0]*len(self.actions),
                index=self.q_table.columns,
                name=state,
            )
        )

```